November 1991

MTP387
Volume 1

Jeffrey Picciotto
Daniel F. Vukelich

Fine Grained Labeling, Volume 1: Operating System Support

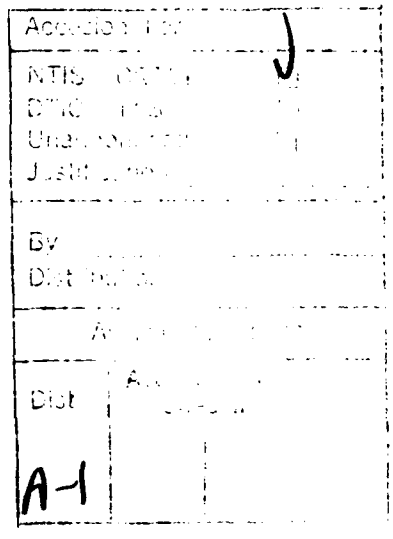
MITRE
Bedford, Massachusetts

November 1991

MTP387
Volume 1

Jeffrey Picciotto
Daniel F. Vukelich

Fine Grained Labeling, Volume 1: Operating System Support



CONTRACT SPONSOR N/A
CONTRACT NO. N/A
PROJECT NO. 028A
DEPT. G111

MITRE

The MITRE Corporation
Bedford, Massachusetts

Department Approval: _____
John P. L. Woodward
Department Head, G111

MITRE Project Approval: _____
Harriet G. Goldman
Project Leader, 028A

ABSTRACT

Trusted computer systems, such as compartmented mode workstations or systems that meet the B-level criteria of the Department of Defense Trusted Computer System Evaluation Criteria, provide a wealth of security-related functionality. In the area of labeling and access control, however, they fall somewhat short. This is because they only provide file-level labeling and access control. Many trusted applications currently envisioned, or under development, rely on a finer granularity of labeling and access control. Such applications include editors that support paragraph markings, message processing systems that label and protect individual messages, and so forth. This paper describes the design and prototype implementation of a general-purpose fine grained labeling and access control mechanism as part of a trusted operating system. The results presented herein indicate that the labeling and access control functionality applications require can be provided in a practical manner with relatively few modifications to the underlying trusted operating system. The resulting benefits to applications, namely reduced size, complexity, and dependence on system-specific security interfaces, suggest that operating system support for fine grained labeling and access control simplifies the design and implementation of such applications while enhancing their portability and minimizing software integration issues.

This page is blank.

TABLE OF CONTENTS

This page is blank.

SECTION 1

INTRODUCTION

The introduction of commercially available trusted computer systems has been a long awaited event. Finally, however, trusted systems such as compartmented mode workstations (CMWs) and systems that meet the B-level criteria in the Trusted Computer System Evaluation Criteria (TCSEC) are becoming available. Users will soon have myriad workstations from which to choose, each of which will support many commercial-off-the-shelf (COTS) applications. To maximize the ability of trusted systems to execute unmodified COTS applications, trusted system developers have tried to minimize the impact on existing applications of the security policies they implement. However, it is inevitable that some trusted applications will nevertheless be required. The reason for this is that trusted systems enforce a variety of security policies, some of which restrict the functionality offered by the system. Applications must be trusted whenever it is necessary or desirable for the applications to enhance, replace, or simply bypass one or more of those security policies. Trusted applications currently under development include trusted mailers and trusted database management systems.

The design and implementation of trusted applications is not merely a matter of selecting arbitrary applications, placing them on a trusted system, and having them operate correctly. Applications must be carefully integrated into the trusted environment provided by the system. For its part, the system must provide certain security capabilities to support the trusted applications. In general, very little research has been performed to determine what security capabilities are required of a trusted system to adequately support trusted applications. Not surprisingly, therefore, trusted systems generally are not designed with the required supporting security features.

To overcome the deficiencies in existing trusted systems, trusted applications frequently implement their own security features. As a result, these applications tend to be less portable (since they often depend on non-standardized security interfaces), and tend to be more difficult to integrate with trusted systems and other applications (since the policies and features the applications implement may differ from those of other applications or those of the trusted system). It becomes evident, then, that in order to maximize application portability and to simplify integration issues, trusted systems should provide as many of the generic security features trusted applications require as possible. An additional benefit is the reduction in size and complexity of the trusted applications as the code implementing the security features is shifted to the underlying system.

Although the security policies enforced in trusted systems cover a variety of functional areas, the labeling and access control requirements have the most substantial impact on applications. Many, if not most, trusted applications are trusted precisely because they enhance the labeling and access control functionality provided by trusted systems. The enhancements that most trusted applications provide generally fall into a single category: increasing the granularity of file labeling and access control. While both the degree to which each application increases the granularity, and the implementation strategy employed, vary

among applications, the fundamental notion underlying these applications is that a fine grained labeling and access control mechanism for files is needed.

A number of cumbersome mechanisms have been developed to address some of the functional limitations imposed by the granularity of labeling available today, however, no general solution has been proposed or implemented. This document summarizes the results of a project to design and implement a general-purpose fine grained labeling and access control mechanism on a CMW base. The results documented here focus exclusively on the fine grained labeling mechanism in order to demonstrate its feasibility and practicality; issues relating to the impact of this mechanism on applications (e.g., architectural changes or simplifications, performance costs, and so forth) are not explored. It is important to note that the implementation presented is a proof-of-concept prototype; it is certainly not the only possible implementation, nor is it necessarily the best. The purpose of the implementation was to gain an in-depth understanding of the design issues and functional tradeoffs of such a labeling and access control mechanism. Finally, although the results presented are discussed in terms of labeling on CMWs, they are applicable to a wide variety of systems and security policies.

This document assumes the reader is familiar with the Department of Defense Trusted Computer System Evaluation Criteria [TCSEC], the CMW requirements given in [CMWREQS], and the MITRE CMW prototype (discussed in [CMWPROTO]) upon which the fine grained labeling and access control mechanism was implemented. This document will therefore neither discuss or explain any of the underlying principles and concepts of trusted systems or CMWs, nor will it discuss the functionality typically offered by standard CMW operating systems or MITRE's CMW prototype.

The remainder of this document is presented in 7 sections. Section 2 addresses the issues that exist without a fine grained labeling and access control mechanism, discusses the goals of fine grained labeling, and defines the particular goals of the prototype implementation. Section 3 provides an informal description of the security policy interpretations enforced by the mechanism. Section 4 provides a brief high-level overview of the architecture used in the prototype implementation. Sections 5 and 6 describe the fine grained labeling and access control implementation in detail. A brief summary of the anticipated impacts on applications due to the mechanism is given in section 7. Finally, section 8 presents the conclusions that can be drawn from this effort.

SECTION 2

FINE GRAINED LABELING GOALS

This section addresses the goals of fine grained labeling (FGL) in two halves. The first half addresses the functional goals. That is, it considers the problems that FGL is intended to address, and the basic functionality that an FGL implementation must provide to solve those problems. The second half specifies the goals, above and beyond the purely functional goals, pursued in the prototype implementation. Clearly there exist many potential architectures for an FGL implementation, however, the possibilities can be narrowed by attempting to achieve a variety of secondary goals (such as extensibility and portability).

Because the functional goals for the fine grained labeling and access control implementation were directly derived from the problems that were intended to be addressed by the mechanism, a review of the issues motivating an FGL implementation is first presented.

MOTIVATION

The CMW requirements given in [CMWREQS] state that, at a minimum, every file must have associated with it a sensitivity level and information label. Sensitivity level-based mandatory access control must therefore be performed at least on a per-file basis. All currently evaluated CMWs implement this minimum, and no more. Furthermore, it is improbable that CMWs completing evaluation in the near future will provide any additional file labeling capabilities. This fact has a critical impact on the operation of many applications. Indeed, most trusted applications are likely to be trusted, at least in part, precisely because this level of granularity for labeling and access control is insufficient to meet their needs.

There are two general classes of applications that need to be trusted as a result of the current approaches to meeting the CMW labeling and access control requirements. The first and most important class consists of those applications that must be trusted because the files they access are shared either by several distinct processes, or by multiple instantiations of a single process, operating at different sensitivity levels. Applications of this type include applications that use well-known files to store shareable data (e.g., dynamic configuration information), as well as virtually all applications that use spool files. This class of applications is critical because the applications that use spool files typically offer services whose absence, or severe performance degradation, users are unwilling to accept. Such applications include mail subsystems, printer subsystems, etc. In fact, this class is considered so important by the trusted system vendor community that less than ideal work-arounds like hidden or multi-level directories have become *de rigeure* and are even included in emerging security standards (e.g., the security extensions to the POSIX interface [SECPOSIX]).

The second class of applications that must be trusted is composed of applications that provide the user with enhanced security features beyond those provided by the trusted system itself. Examples of these types of applications include databases that provide per-record labeling

and access control, editors that support paragraph markings, and message processing systems capable of managing the labels on, and access to, individual messages (or portions of messages, if support for the transmission of documents containing paragraph markings is desired). Few applications of this type exist today. However, it can reasonably be anticipated that as trusted systems become more prevalent, and users gain a better grasp of the systems' capabilities and limitations, the demand for trusted applications that provide the most convenient, flexible, and powerful security features will increase.

The common thread underlying the two classes of applications described above is that the applications in both classes need to associate labels with, and enforce mandatory access control on, objects typically kept within a single file. Thus, for example, a trusted mail subsystem needs to store and control access to messages possessing different sensitivity levels in a single spool file, and a trusted editor needs to associate different labels with each paragraph of text within one file. Applications are currently being developed that provide their own mechanisms for independently labeling various portions of files and that enforce the system's mandatory access control and information label floating policy on access to the files.[1] This type of application-level solution to a global problem suffers from several severe drawbacks. The most notable drawbacks are as follows.

Overuse of privileges. The highly trusted code necessary to implement the requisite MAC checks must be replicated in each application. This essentially guarantees that each application must possess what are likely to be the most powerful privileges available on the system. It would be preferable to more stringently adhere to the least privilege principle and minimize the use of such highly trusted code, particularly in commercially developed application level software.

Duplication. The trusted code needed to manage the sensitivity and information labels must similarly be replicated in every application. Note that the code required to store, retrieve, and update the labels as portions of the file are accessed or modified is likely to be quite complex and rather substantial.

Limited interoperability. Interoperability between applications is unlikely: each application developer will undoubtedly devise a unique mechanism and data format to implement fine grained labeling. No application's mechanism is likely to be compatible with any other application's mechanism. Even the details of the security policies are liable to differ in slight, but potentially critical, ways. Furthermore, since access to an application's file cannot be limited to interfaces provided by that application (unless all access is limited), the data in the file must remain protected by a high water mark label that the system uses to mediate access by untrusted processes. This significantly reduces the potential benefits of fine grained labeling.

---

[1]The only operating system support that has ever been developed is "multi-level" or "hidden" directories. However, both these solutions are rather inelegant mechanisms that solve only a limited subset of the general problem. That is, they only solve the problem of name space collisions when multiple files with different labels share the same name.

To provide the granularity of labeling and access control that many applications desire in a manner that addresses the issues listed above, the FGL mechanism described in this paper was developed. The general functional goals of an FGL mechanism are discussed in the following subsection.

MINIMUM FUNCTIONALITY

The intent of an FGL mechanism is to shift all the trusted code necessary to maintain and manipulate labels from within the various trusted applications to a single operating system subsystem. The granularity and labeling goals, described below, ensure that the burden of storing and maintaining the labels is removed from the applications. The policy goal ensures that the burden of policy enforcement is similarly shifted from trusted applications to the FGL operating system subsystem. In many cases, meeting these goals immediately eliminates the concerns listed above. The application support and compatibility goals simply try to guarantee that the implementation of an FGL mechanism does not eliminate existing functionality.

Note that these goals specify the minimum functionality an FGL implementation must provide in order to adequately address the concerns listed above. Commercial implementations could choose to provide additional features.

Objects. The mechanism must be applicable to files on a per-file selectable basis (i.e., an appropriately authorized user must be able to specify which files are FGL and which are not). The term files includes regular files, and is not required to include other file types (e.g., UNIX[2] device special files).

Labeling. The labeling must include both information labels and sensitivity levels.

Granularity. The degree of granularity must match the granularity of the object. In the case of UNIX files, where file operations are byte-oriented, this implies that each byte must be individually labeled (either implicitly or explicitly).

Policy. The mechanism must implement all the necessary mandatory access control checks, information label floating, privilege checks, and auditing needed to satisfy the CMW requirements. A discussion of the precise interpretation of the applicable security policies, being somewhat complex, is deferred until section 3.

Application support. Some trusted applications will require direct access to the fine grained labels in order to implement extended labeling enhancements. Therefore, trusted interfaces to determine and manipulate the labels must be provided. In addition, mechanisms for untrusted applications to determine the label of the data they successfully read from an FGL file must be provided.

---

[2]UNIX is a trademark of AT&T Bell Laboratories.

Backward compatibility. To prevent the FGL mechanism from being more of a hindrance than a help, the mechanism must be designed such that applications that functioned correctly within the CMW security constraints prior to the introduction of FGL files should continue to function correctly, without modification, in the presence of FGL files.

DESIGN GOALS

The prototype also attempted to achieve several specific design goals. While these goals, listed below, do not directly impact the primary functionality offered by the FGL mechanism, they do affect the implementation's architecture and are therefore sufficiently significant to warrant consideration.

Extensibility. The FGL mechanism is intended to provide for the fine grained application of labels to the data within files. Nevertheless, the mechanism should be sufficiently flexible to support the fine grained application of other attributes. Examples of other attributes include discretionary access control information, object type information (see [OBJTYPE] for a discussion of object typing), or any other data attributes required by security policies enforced by the system.

Extensibility applies to the environments in which the FGL mechanism can be used. Clearly the mechanism must be applicable to any local file. With the growing use of distributed data management (e.g., using NFS or networked databases), there is an increasing need for security policies to be applied to file accesses which occur on systems other than that from which the file access request originated. The FGL mechanism should be able to support these types of distributed data accesses.

Simplicity. The focus of proof-of-concept prototypes is on correctly implementing the basic functionality desired; advanced, or merely convenient, features are typically omitted. The prototype implementation described below strictly adheres to this philosophy.[3] Nevertheless, the basic architecture and the fundamental design choices must not be compromised. Thus, while there is no obligation to provide a wealth of sophisticated interfaces and features based on complex yet efficient algorithms, it is important that the design and implementation be capable of supporting such features.

Portability. The benefits of an FGL mechanism would clearly be substantially increased if CMWs in general supported FGL files. As noted earlier, most CMWs under development do not currently provide any FGL mechanism; such a mechanism would therefore need to be retrofitted to an existing base. To simplify such retrofitting, the FGL design should be as portable as possible.

Performance. Performance, in a proof-of-concept prototype, is only critical when there is doubt that adequate performance is achievable. Popular opinion tends toward the belief that an FGL mechanism can never achieve acceptable performance. In this

[3]The implementation is less than production quality. For example, the interfaces are unsophisticated, algorithms were selected for clarity and simplicity rather than impressive performance, and so forth.

case, therefore, adequate performance must be provided. However, implementing highly efficient, optimized, code is frequently a long and arduous task which provides little benefit in the context of a prototype. The implementation must therefore walk a fine line: although it need not provide as much performance as theoretically possible, it should provide enough performance to make a compelling argument that a fully optimized implementation would provide an acceptable level of performance.

This page is blank.

SECTION 3

SECURITY POLICY

This section describes the security policy interpretations enforced by the prototype fined grained labeling (FGL) mechanism. The particular interpretations described in this section were designed to maximize simplicity and ease of use from the application perspective, occasionally at the cost of flexibility. Other, equally valid, interpretations are possible.

SCOPE OF POLICY

The FGL implementation enforces a mandatory access control (MAC) policy and an information labeling (IL) policy. The mechanism does not enforce discretionary access control, object typing, or any other security policy; these policies remain enforced by the CMW on a per-file basis. The FGL mechanism was implemented in such a fashion that retrofitting additional policies would be a simple task requiring no modifications to the FGL architecture.

The rationale for implementing only MAC and IL policies is based on the observation that MAC and IL are the two most prevalent policies that lead to the types of problems discussed in section 2. Thus, by implementing these policies within the FGL mechanism, the prototype serves as a complete proof-of-concept: it demonstrates that a useable FGL implementation can be achieved and that the primary issues it was intended to address were, in fact, resolved by that mechanism. As noted above, the architecture of the FGL mechanism (described in later sections) has been carefully designed to permit new security policies to be introduced into the mechanism. Therefore, should future functional requirements indicate a need for the application of other policies at a finer granularity (for example fine grained integrity controls or fine grained discretionary access control), these can be accommodated without completely redesigning the existing software.

BASIC SECURITY POLICY

The details of the MAC and IL policy interpretations are presented below. For the purposes of simplifying the policy discussion, the operations permitted on files by UNIX are grouped into six classes: read, write, append, seek, truncate, and get length. At a purely theoretical level, some of the classes are hierarchically related to other classes (e.g., write includes append), however, the FGL prototype treats each class somewhat differently. Thus, by examining the cases independently rather than as a generalized aggregate, the policy becomes clearer. Note that these six classes are sufficient (either individually or in combination) to support all UNIX file operations relevant to an FGL mechanism.

The discussion below assumes that a process is attempting the operation being described on a file available for access. That is, the discussion does not address the policies that are applied on a per-file basis (e.g., discretionary access control and object typing). Furthermore, FGL

files are assumed to consist of arbitrary lengths of individually labeled bytes. Assuming byte-level labeling is appropriate for the prototype because, as previously discussed, UNIX operations on files operate on data at the byte level. However, the discussion can equally well be applied to different labeling granularities (such as bit-level labeling).

Mandatory Access Control

The MAC policy interpretation adopted by the FGL prototype adheres to, yet is more restrictive than, both the Bell-La Padula model [Bell] and the CMW MAC policy specified in [CMWREQS]. Specifically, write operations are more tightly constrained than required. The general notion, more explicitly stated below, is that processes running with different sensitivity levels may have different views of a file depending on the sensitivity levels of the data within the file. In general, processes are permitted to view (e.g., read and seek) bytes whose sensitivity level they dominate. Processes may overwrite bytes whose sensitivity levels equal the processes' sensitivity level.

Obviously, a write equal policy is enforced. Writing down is not permitted because it violates the Bell-La Padula model. Writing up is not permitted because it was deemed extremely confusing to applications. Specifically, the FGL mechanism is intended to present a coherent view of a file. Allowing a process to overwrite bytes that it cannot read, and whose existence cannot be known to the process, is not consistent with this goal. In fact, it may lead to confusing and unexpected results. This confusion is more profound than in the traditional case of writing up to a file. In the case of a file, the application is generally aware that it is writing up. At very least, it can determine nothing about the file, and is therefore unlikely to be modifying the file based on the file's existing contents. In the case of FGL files, a process may read the file then rewrite a slightly modified version of the data. If write up were permitted, writing the modified data may result in the inadvertent deletion of higher level data. Untrusted processes would have no means to avoid this problem, nor even to determine when it might arise.[4] For these reasons, write up is not supported.

Note that in some environments there may exist applications that could benefit from the ability to write up to higher level portions of FGL files. Therefore, a maximally flexible MAC policy should provide a write up capability. However, because such applications are likely to be quite rare, the default MAC policy for general purpose systems should probably remain write equal.

The following list specifies the precise policy interpretation for mandatory access control. Note that the standard error conditions (e.g., attempting to seek beyond the end of a file or attempting to read more bytes than are available) still generate the same operating system-specific error return values, although these are not described below. New error conditions have been added only in cases where a process attempts to delete (or overwrite) lower level data.

---

[4]Informing an untrusted process that higher level data was present would introduce a covert channel and is therefore undesirable in trusted systems.

Read A process that attempts to read *N* bytes of data is returned the first *N* bytes, starting at the current process offset into the file, whose sensitivity levels (SL) are dominated by the process SL. Bytes whose SL are not dominated by the process SL are silently ignored.

Write When a process attempts to write *N* bytes to an FGL file, any bytes at the current process offset into the file whose SLs dominate, but do not equal, the process SL, are silently ignored. If any of the *N* bytes starting at the process offset that are dominated by the process SL are not equal to the process SL, then the write fails.[5] Otherwise, the *N* bytes whose SLs equal the process SL are overwritten.

Append The new bytes are appended to the end of all existing bytes in the file (regardless of their SL). The new bytes are labeled with the process' SL.

Seek A process that attempts to seek to the *N*th byte offset is placed at the position in the file where the sum of the number of preceding bytes whose SL is dominated by the process SL is *N*. Preceding bytes whose SL are not dominated by the process SL are silently ignored.

Truncate A process that attempts to truncate a file to length *N* effectively truncates the file to length *N* with respect only to those bytes whose SLs are dominated by the process SL. Thus, an automatic seek to byte *N* (as described above) is performed. If any of the remainder of the bytes in the file are dominated by, but are not equal to, the process SL, the truncate fails. If all remaining bytes have SLs that dominate the process SL the truncate proceeds as follows. All remaining bytes whose SLs equal the process SL are deleted. All remaining bytes whose SLs dominate the process SL are silently ignored.

GetLength The length returned is the number of bytes whose SL is dominated by the requesting process' SL. Bytes whose SL are not dominated by the process SL are silently ignored.

Information Labeling

The information labeling policy interpretations are specified below.

---

[5]The lower level bytes are not silently ignored because of the confusion that would result: applications would be able to view, but not modify, portions of the file. Rather than silently skipping over the lower level data and overwriting later data that the application could, but perhaps did not intend, to modify, an error is returned. In this way, applications are alerted that the action they most probably desired could not be performed. FGL-cognizant applications should be able to recover from the error. FGL-ignorant applications must have intended the lower-level data to be overwritten, so the error is appropriate.

Read The information label of the reading process is floated with the information label of every byte returned by the read operation.

Write The information label of the bytes written are set to the process information label at the time the write occurred.

Append The information label of the bytes written are set to the process information label at the time the write occurred.

Seek No information labeling actions occur.

Truncate No information labeling actions occur.

GetLength No information labeling actions occur.

PRIVILEGES

As expected, the policies described above apply to all untrusted applications. Trusted applications possessing appropriate privileges may bypass these policies. The prototype FGL implementation supports the privileges described below.

PRIVILEGE	PURPOSE
READ_UP	Bypass MAC checks on read, seek, and get attribute operations.
WRITE_DOWN	Bypass MAC checks on write and delete operations.
NO_AUTO_PSET_SEC_LBL	Prevent process IL floating on reads.
FGL_OVERRIDE	Bypass entire FGL mechanism.

Table 1. List of Supported Privileges

Note that there is no mechanism to bypass the automatic labeling of data on write and append operations on FGL files. This is because all data within an FGL file must be labeled. However, trusted applications may use the FGL_OVERRIDE privilege to bypass the entire FGL mechanism and directly access and modify FGL files (and their associated security information). In a more complete implementation, it is likely this single privilege would be replaced with several less all-encompassing privileges.

SECTION 4

ARCHITECTURAL OVERVIEW

This section provides a high level overview of the architecture of the prototype fine grained labeling (FGL) mechanism.

SCOPE

The FGL mechanism, as implemented, applies only to regular UNIX files. All other objects (such as devices, symbolic links, FIFOs, and windows) are not subject to the FGL policy. The rationale for this limitation is that in order to successfully resolve the issues outlined in section 2 and implement a convincing proof-of-concept prototype, addressing objects other than files is unnecessary. In addition, in many cases it appears improbable that applying the FGL mechanism would be useful. Supporting FGL devices and symbolic links, for example, would not seem to be a worthwhile enhancement. In some cases, the underlying CMW operating system upon which the mechanism is implemented already provides the FGL functionality. Pipes and sockets, for example, are effectively subject to the FGL policy since the CMW base includes a trusted socket implementation (described in [TSOCK]).

ARCHITECTURE

MITRE's original CMW prototype embedded mandatory access control (MAC) and information labeling (IL) policy code directly into the CMW's UNIX kernel. Because the original prototype implemented MAC and IL policies on files in a straightforward manner, the amount and complexity of code was small. Thus, embedding that code in the kernel was a practical and effective approach. In contrast, the FGL policies that must be enforced, being more intricate, require more code of substantially increased complexity. As a result, embedding the FGL code in the kernel was not deemed wise.

The size and complexity of the FGL implementation introduces several drawbacks to embedding the FGL code in the kernel.

> *Implementation.* Programming errors in code embedded in a UNIX kernel tend to result in catastrophic system failures rather than trivial program failures. Furthermore, debugging a UNIX kernel is neither a simple nor a rapid process.
>
> *Enhancement.* It was anticipated that the originally-envisioned FGL policies would be refined as the implementation progressed. Enhancing a UNIX kernel is a slow and tedious process compared to updating an application.
>
> *Portability.* Code embedded in a kernel is not easily ported to different operating systems; code that exists as a user-level application that depends on only a few hooks in a kernel is far more portable. In addition, application-level code that is insulated

from the precise workings of the underlying kernel can be accessed via library routines by trusted applications and, in this way, used on non-CMW systems where the FGL functionality is still desired.

Thus, to maximize ease of implementation, enhancement, and portability, the FGL functionality was implemented as a trusted user-level process called a daemon. The figure below illustrates this architecture. Referring to the figure, in step 1 an arbitrary user process performs a system call for which FGL mediation is required (e.g., reads an FGL file). In steps 2 and 3, the kernel invokes the FGL daemon and awaits its response. Based on the daemon's response, in step 4 the kernel returns success or failure of the system call (and, if successful, any data to be returned) to the user process.

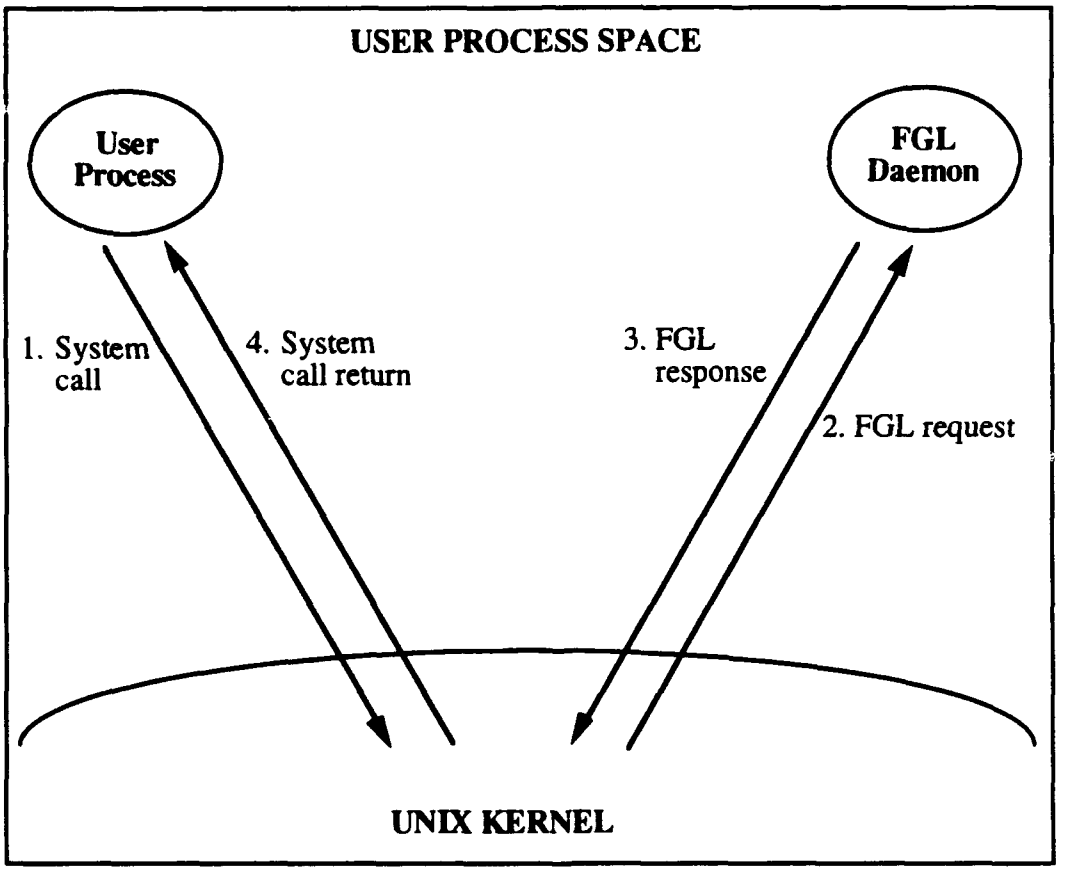


Figure 1. FGL Architecture

The determination of whether or not the FGL daemon must be invoked for a particular file operation is based on the attributes of the file. This permits arbitrary files to be selectively designated as FGL files. FGL files are identified as such to the system by a reserved object

subtype (see [OBJTYPE]). As a result, the code embedded in the kernel is neither extensive nor complicated. When a system call is performed, it merely checks the file attributes and invokes the FGL daemon only if necessary.

Given the architecture above, the heart of the FGL prototype implementation can be broken into two distinct components. The first component is the FGL protocol used by the kernel to communicate with the FGL daemon. The design and implementation of this protocol is discussed in the next section. The second component is the FGL daemon itself. The design and implementation of the daemon is described in the subsequent section.

This page is blank.

SECTION 5

PROTOCOL IMPLEMENTATION

The purpose of this section is to present the FGL protocol implementation and show how this implementation meets the functionality, design, and architectural goals presented earlier in this paper.

GOALS AND OBJECTIVES

In addition to the general goals presented earlier, some specific goals and objectives were identified with respect to the implementation:

> *Implementation Independence.* Specify only the *functional* requirements of the FGL procedures, thus liberating the specification from any implementation dependencies. Also, represent procedural data (i.e., input and output parameters) in a machine independent language, thus fostering interoperability among various machine architectures.
>
> *Robustness.* Design the FGL system so that it is possible to recover gracefully in the event of a daemon crash. Also, consider efficiency; at the granularity of FGL procedures, this means minimizing calls to the daemon, since calls to the daemon may result in several context switches.
>
> *Standards.* Specify the FGL procedures in a standard data representation language, and implement the FGL procedures using standard communication mechanisms.

IMPLEMENTATION

The following paragraphs present the implementation details regarding both kernel support for FGL and kernel/daemon communication; details regarding the daemon implementation follow in the next section.

The Client/Server Architecture

As noted in section 2, one of the functional goals in implementing the FGL system was to maintain backward compatibility with existing applications that use the file system. Ultimately, *binary* compatibility is desirable, since it does not even require that applications be recompiled much less modified in order to operate in the new environment. In order to achieve binary compatibility with existing applications, it is necessary to re-implement every system call that involves file access (e.g., **lseek**) in terms of FGL files, and at first it may seem that a straightforward approach would be to implement each system call as a procedure call to the FGL daemon. However, since it is expected that calls to the user-level FGL daemon will be relatively expensive (in that it may involve several context switches), it is

wise to avoid unnecessary calls to the daemon. Fortunately, a more effective solution is available, and this solution is based upon some observations that can be made regarding how file access is currently implemented in the UNIX kernel.

The file-access system calls manage two distinct states: the *per-process* state, which contains information such as the process' current offset into an open file, and the *file* state, which contains both the file data and the attributes associated with that data, such as file size. Moreover, these system calls do not manage file state directly but rather indirectly through a set of interface procedures provided for file management. What follows from this abstraction is an FGL implementation that satisfies the architectural goals: As before, the system calls invoke the interface procedures to manage file state and maintain responsibility for managing per-process state; however, in the case of FGL files, the interface procedures are implemented in the daemon, and procedure invocation is accomplished through message exchanges between the kernel and the daemon.

Besides satisfying our architectural goals, the added benefits of this file management abstraction are as follows: First, the specific access policies that govern file manipulation are hidden from the more general system calls. Second, as will be shown, maintaining responsibility in the kernel for managing per-process state rather than shifting the burden to the daemon simplifies recovery in the case of a daemon crash.

It is instructive at this point to consider some examples that demonstrate the functionality of the architecture. Suppose that an application invokes the following system call:

lseek (fd, 0, L_SET);

The effect of this system call is that the process' offset for the open file identified by **fd** is set to the value indicated in the second parameter, which in this case is **0**. The semantics of this call affect per-process state but not file state; hence, it is not necessary to invoke any FGL procedures and incur the concomitant overhead of communicating with the daemon.

On the other hand, suppose that an application invokes the same system call as above, but uses the **L_XTND** parameter in place of the **L_SET** parameter. The effect of this system call is that the process' offset for the open file identified by **fd** is set to the size of the file *plus* the value indicated in the second parameter, which in this case is **0**. As before, the per-process state is affected by the operation in that the offset into the file is updated; however, in this case it is necessary to know the size of the file, which is maintained in the *file* state. Hence, it is necessary to communicate with the daemon in order to interrogate the file size.

FGL File Procedures

What follows is a high-level description of the primitives for managing FGL files in the daemon. It should be noted that the requirements of all system calls for manipulating FGL files can be implemented by some particular sequence of the primitives defined here. A more formal definition of these primitives appears in the appendix. The primitive routines are as follows:

FGLPROC_OPEN (fglopenargs) return fglstat;

This call announces to the FGL daemon that it should initialize the file for subsequent accesses. "fglopenargs" contains the following information: the absolute pathname of the file to open; a file handle, "fglfhandle", which will serve as a synonym for the pathname in subsequent access requests; and the security attributes associated with the application making the open request. Upon return, "fglstat" contains the status of the request (i.e., success or failure).

FGLPROC_WRITE (fglwriteargs) return fglstat;

This call requests that the FGL daemon store the sequence of bytes provided by the application within the FGL file. "fglwriteargs" contains the following information: "fglfhandle", indicating which file to write; the number of bytes of data to write; the data itself; the position in the file at which to start writing; and the security attributes associated with the data to write. Upon return, "fglstat" indicates whether the request succeeded or failed. Note that this specification meets the granularity requirement for labeling identified in section 2 in that the amount of data written may be limited to a single byte.

FGLPROC_READ (fglreadargs) return fglrdresult;

This call requests that the FGL daemon retrieve a sequence of bytes within the FGL file and return these bytes to the application. "fglreadargs" contains the following information: "fglfhandle", indicating which file to read; the number of bytes of data to read; the position in the file from which to start reading; and the security attributes associated with the application at the time it issues the read request. Upon return, "fglrdresult" provides the following information: an indication whether the request succeeded or failed and, if the read succeeded, the data read, a byte count, and the security attributes associated with the data returned. (The byte count is necessary since the actual number of bytes read may be less than that requested). The security attributes returned with the data are used to enforce any access policies that may apply to the application as the result of having read the file. Note that this specification meets the granularity requirement for labeling identified in section 2 in that the amount of data read may be limited to a single byte.

FGLPROC_GETATTR (getattrargs) return fglattrstat;

This call requests that the FGL daemon return the attributes associated with the file. "getattrargs" contains the following information: "fglfhandle", indicating which file to query, and the security attributes associated with the request. Upon return, "fglattrstat" contains the following information: an indication whether the request succeeded or failed and, if the request succeeded, the attributes associated with the file, which include security attributes. The security attributes returned are used to enforce any access policies that apply to the application as the result of having queried the file.

FGLPROC_SETATTR (setattrargs) return fglstat;

This call requests that the FGL daemon set the attributes associated with the file to those specified by the requesting application. "setattrargs" contains the following information:

"fglfhandle", indicating which file to update; the file attributes to apply; and the security attributes associated with the application issuing the request. Upon return, "fglstat" indicates whether the request succeeded or failed.

FGLPROC_CLOSE (fglfhandle) return fglstat;

This call announces to the FGL daemon that it should close the file, since access is no longer required. "fglfhandle" indicates which file to close and, upon return, "fglstat" indicates whether the request succeeded or failed.

Integrating the FGL Primitives with the System Calls

To recap some points made earlier, applications do not invoke the FGL procedures directly; rather, the application invokes the standard system calls for file manipulation (e.g., **lseek**), and the system calls in turn invoke the FGL procedures. Moreover, the requirements of the system calls for manipulating FGL files can be implemented by some particular sequence of the primitives defined in the preceding section. The following example illustrates these points. Suppose that an application invokes the following system call:

lseek (fd, 0, L_XTND);

If the file descriptor, **fd**, refers to an FGL file, the **lseek** system call invokes the FGLPROC_GETATTR() routine, which returns the size attribute associated with the file. **Lseek** can then update the per-process state by adding the value of the size attribute returned from the daemon to the value indicated in the second parameter, which in this case is **0**, thus preserving the semantics of the routine.

It is interesting to note that there are really two classes of system calls that involve file access: those that access a file by *file descriptor* and those that access a file by *name*. In the first class, the application explicitly opens the file and is returned a file descriptor from the kernel by which the application then accesses the file in subsequent operations; processing of the file (usually) concludes with an explicit close operation, which relinquishes use of the file descriptor. By contrast, the routines that access files by name require no explicit open and close of the file on the part of the application in order to perform the requested file operation.

In the current specification of the protocol, the use of file handles between the kernel and the daemon is comparable to the use of file descriptors between the application and the kernel: the kernel explicitly invokes the FGLPROC_OPEN() on behalf of the application, which establishes the file handle for use in the subsequent FGL procedures; typically, processing of the FGL file concludes with an explicit FGLPROC_CLOSE() operation, which relinquishes use of the file handle. In the current implementation this means that the second class of system calls must be accomplished by a sequence of procedure calls to the FGL daemon: explicitly open the file, perform the desired operation, and explicitly close the file.

Remote Procedure Call

Communication between the kernel and the daemon was implemented using Sun Remote Procedure Call (RPC) [RPC]; that is, the FGL procedures given above were converted to

remote procedure calls in which the kernel invokes the procedure, and the daemon implements the procedure and returns a result. The reasons we selected this mechanism are as follows: First, it is a standard communication mechanism available to both kernel-based routines (through kernel-RPC) and user-level processes (through RPC libraries). Second, it is straightforward to generate RPC language directly from the formal definitions given below. Third, one of the design goals identified in section 2 was simplicity, and using RPC meant that no additional effort had to be expended on developing an ad hoc mechanism for kernel/user communication.

It is important to note, however, that the FGL specification does not depend upon, nor dictate use of, RPC; rather, any suitable interprocess communication (IPC) mechanism (sockets, shared memory, special devices, etc.) can be substituted without loss of generality in the specification.

Pathnames and File Handles

Given the architecture, it is necessary for the kernel and the daemon to agree upon a unique name for referencing the FGL file, and absolute pathnames serve this purpose: absolute pathnames uniquely identify files, and these names can be used by both the kernel and user-level processes in referencing files.

The problem is that when an application opens a file, the pathname specified may not be *absolute* but *relative to the current working directory*, and the current working directory of the application may not coincide with the current working directory of the daemon. This means that, in general, when an application opens a file, it is necessary to resolve the pathname (i.e., convert it to an absolute pathname) before passing the name to the daemon. Moreover, each FGL operation that the kernel invokes must somehow indicate the file to which the operation applies; as such, once an absolute pathname is agreed upon between kernel and daemon, this pathname could be used in all subsequent operations the application requests.

The problem with using absolute pathnames is that each pathname may consume as much as 1024 bytes of storage, which poses a potential storage problem and increases communication overhead as the number of applications and open files multiplies. The solution was to define a "file handle", which is a shorthand name for referencing files. Hence, when the kernel opens an FGL file on behalf of the application, it provides the daemon with (1) an absolute pathname, which allows the daemon to unambiguously identify the file the application wishes to reference, and (2) a file handle to associate with that pathname, which will be used by the kernel in subsequent requests as a shorthand notation for the absolute pathname.

In the kernel, a pathname refers to a file that is also uniquely identified by these attributes: (*file system, inode-number, generation-number*). We exploit the uniqueness of these attributes by defining a "file handle" that contains only this ordered triple and by using this handle as an abbreviation for the pathname. Thus, the file handle used in the prototype implementation is as follows:

```
struct fglfhandle {          /* File Handle Definition */
    long        fid[2];      /* File System Identifier */
    u_long      ino;         /* File Inode Number */
    long        gen;         /* File Generation Number */
};
```

Not only is this identifier small, but also its components can be easily and quickly derived whenever the application references the open file; hence, no additional storage is required per process.

Usage Counts

Since resolving absolute pathnames is currently an expensive operation, further efficiency was achieved in the prototype implementation by observing that as soon as one application succeeds in opening a file–thus announcing to the daemon the absolute pathname and the file handle that will be used to reference the file in subsequent operations, another application which opens the same file concurrently need not perform the "opening" protocol with the daemon, since the shorthand use of the file handle has already been established by the first application. Likewise, if an application closes a file while other applications still have the file open, the application need not perform the "closing" protocol with the daemon; only the last application to close the file is required to announce to the daemon that the file handle will no longer be referenced. Note that the "closing" protocol is important because it allows the daemon to free resources associated with that file handle.

The functionality just described was implemented by means of a *usage count* associated with each file node. When an application opens a file, if the usage count on the file node is zero, this indicates that the pathname and its associated file handle must be announced to the daemon; otherwise, if the usage count on the file is greater than zero, no "opening" protocol is necessary; the usage count is simply incremented. When an application closes a file, if the usage count on the file node is zero, this indicates that the daemon must be informed that the file handle is no longer referenced; otherwise, if the usage count is greater than zero, no "closing" protocol is necessary; the usage count is simply decremented.

Note that the data structures described above (pathnames, file handles, and usage counts) are all maintained by the trusted operating system and are unavailable to application software. They therefore do not introduce covert storage channels. However, usage counts may introduce timing channels since certain file operations (such as open and close) will probably operate much faster if the file referenced is also open by another process. There are likely to exist a wide variety of other timing channels in the implementation described in this document. No covert channel analysis was performed on the FGL design and implementation as such an analysis was considered outside the scope of the project.

Support for Security Policies

In many cases, the FGL procedures contains security attributes, and these security attributes can be used to enforce security policies. In the next paragraphs the FGL support for security policies is discussed in terms of policy enforcement and applications programming interfaces.

Policy Enforcement

In the FGL architecture, it is the responsibility of the daemon to enforce security polices on the file objects the application attempts to access, and it is the responsibility of the kernel to enforce security policies that pertain to the application. In order to support this requirement, the kernel must associate security attributes with the requests it sends to the daemon, and the daemon must associate security attributes with the replies it returns to the kernel.

It should be noted that the definitions of the primitives given earlier did not specify exactly what policies apply nor how these policies are enforced. In other words, security attributes are treated as opaque data with respect to the protocol, thus satisfying the extensibility requirements identified in section 2. Nevertheless, the prototype implementation incorporated CMW security attributes in the FGL protocol and enforced the policies associated with these security attributes in both the kernel and the daemon. The following paragraphs describe the kernel support for CMW policies; the daemon support is presented in the next section.

The kernel provides a sensitivity label, information label and, in the case of trusted processes (see below), a set of privileges with each FGL procedure it invokes, thus satisfying the labeling requirement identified in section 2; these attributes are used by the daemon to enforce CMW policies on the file objects the application is attempting to access.

The responses to the FGLPROC_READ() and FGLPROC_GETATTR() received from the daemon contain the security attributes associated with the information returned; the kernel uses these attributes to enforce the appropriate CMW policies, thus satisfying the policy requirement identified in section 2. MAC policy is actually not enforced by the kernel on these responses, since the daemon has already adjudicated the access; however, the information label returned with a response *will* be used to float the information label associated with the application making the request.

Application Support

Although access policies pertain to all *untrusted* applications of the FGL system, *trusted* processes may need to override these policies, thus requiring an appropriate privilege. Of particular interest is the FGL_OVERRIDE privilege; it is this privilege which allows the FGL daemon to access FGL files directly rather than indirectly through the FGL daemon, thus avoiding an infinite recursion.

In order to meet the application support requirements presented in section 2, it was necessary to provide an applications interface for determining–and in the case of trusted processes, manipulating–the security attributes associated with file access requests. Rather than introducing new system calls for this purpose, new functionality was added to the existing **open**, **read**, and **write** system calls.

The **open** system call already includes a *flags* parameter that allows certain options to be specified when the file is opened. For example, the "O_CREAT" flag specifies that the file is to be created if it does not exist. To this set of flags, the "O_FGL" flag was added, which has the following meaning: When the file is opened using this flag, the kernel associates this flag

with the file descriptor it returns to the user; subsequently, whenever the application references this file descriptor in a **read** or **write** call, the kernel interprets the call according to the rules specified for FGL, which are described in the following paragraph.

The **read** and **write** system calls each specify four (4) parameters: *fd*, a file descriptor, which is the handle by which the application accesses a file; *buf*, a buffer, which contains the data to be read or written; *len*, a length parameter, which indicates the amount of data contained in the buffer; and *flags*, a set of flags that apply to this read or write. If the "O_FGL" flag is associated with the file descriptor referenced in one of these calls, the kernel interprets the buffer parameter as follows: On **read**, the kernel supplies the security attributes associated with the data read in the buffer, beginning at offset *len* + 1; that is, the security attributes can be read by the application beginning at *buffer*[*len* + 1]. On **write**, the application supplies the security attributes associated with the data in the buffer, beginning at offset *len* + 1; that is, the security attributes can be extracted by the kernel beginning at *buffer*[*len* + 1]. Of course, in order for this new interpretation to succeed, the application must allocate buffers large enough to hold the maximum amount of data to be read or written plus the size of a security attributes structure.

Crash Recovery

In the FGL environment, it is possible that either an application or the FGL daemon may crash while an FGL file is open. In the prototype implementation, handling an application crash is straightforward: when the application "exits" the FGL file is closed. If this is the last application to close the file (see the discussion above regarding "usage counts"), the daemon is notified so that it may liberate any resources it has associated with the file; if this is not the last application to close the file, other applications may continue to access the file normally.

A daemon crash, on the other hand, is serious because it affects many applications; that is, no application can access the file until the daemon recovers. In the current implementation, the kernel simply notifies the application of any instabilities perceived in communicating with the FGL daemon (timeouts, stale file handles, etc.) and frees any per-process state associated with the open file; hence, the application is required to re-open the file in order to resume access. The purpose of a robust crash recovery scheme is to hide these system instabilities–and therefore the FGL implementation details–from the application, thus allowing the application to deal exclusively with file access errors. In other words, insofar as the application should be concerned, file access requests never timeout, and file handles do not even exist much less become stale. Although the current implementation does not provide such robustness, the following paragraphs sketch a potential solution.

When the daemon crashes, it loses any state information it was maintaining in memory; however, the only state information that the daemon maintains with respect to applications is the mapping between file handle and pathname, and the kernel can reiterate this mapping, if necessary. Thus, the only adverse effect a crash should have upon an application is an increased latency on file operations. As a consequence of this, RPC failures, which erstwhile were reported to the application upon failure, never fail but are retried infinitely until the daemon responds and the file handle is refreshed; that is, the application remains blocked until its file access request can be satisfied.

Recovery begins when the daemon is again willing to accept requests–at which point it can safely be assumed that the daemon has no state; that is, the daemon has no knowledge of any active file handles nor the pathnames to which these handles apply. Hence, when an operation the kernel has been retrying on behalf of an application arrives at the daemon, the daemon is presented with a file handle it does not recognize; whenever this situation occurs, the daemon returns an error indicating that the file handle is stale. In the current implementation, this error is delivered to the application–a situation that may prove quite mortifying; in the proposed scheme, the error returned informs the kernel that it should reiterate the mapping between file handle and pathname (by performing the FGLPROC_OPEN() procedure). Once the mapping is reestablished, the kernel can then continue with the operation it was attempting to perform when it was notified that the file handle had become stale.

The process of refreshing the stale file handle is as follows: Because many applications may be simultaneously retrying requests that have failed due to communication problems, when the daemon resumes operation it may be bombarded with requests containing an unrecognized file handle, in which case the daemon will return an error indicating that the file handle is stale. Nevertheless, as soon as one of these errors is returned, the kernel, acting on behalf of the application that would normally receive the error, can commence recovery. When the error is received, the kernel marks the file node as "opening" and pathname resolution is executed. While in this state, any application that attempts to access the file is suspended until the "opening" protocol completes; likewise, any responses received from earlier attempts to contact the daemon with a stale file handle results in queueing up the application for access when the "opening" protocol completes. Finally, when the "opening" protocol completes, all pending applications are awakened.

Hence, crash recovery is similar to the normal situation that occurs when several applications request to open the same file: one application participates in the "opening" protocol, and the other applications are suspended until the "opening" protocol completes; the only difference between the two is that the normal situation results in an update of the usage count, whereas usage counts are not affected during crash recovery (unless, of course, existing applications exit or new applications attempt to open the file).

FGL PROTOCOL

The FGL protocol is a set of procedures invoked by clients to access FGL files. It is important to note that the FGL procedural definition does not specify any particular implementation; the definition only states the *functional* requirements of the protocol and is therefore implementation independent. It is also important to note that the procedural definitions that follow are specified in an abstract syntax, thus eliminating any machine dependent representation of the data structures.

The procedures of the FGL protocol are assumed to be synchronous: when a procedure call returns to the application, the application can assume that the operation is complete. For example, when the kernel receives the response message from a write request, the kernel can assume that the data associated with the request has been written and that any applicable file

attributes have been updated (e.g., file size). The precise protocol definition, given in the External Data Representation Standard (XDR) [XDR] notation, is provided in the appendix to this document.

FUTURE WORK

In the prototype implementation of FGL, security attributes are included in the definitions of FGL messages; that is, objects (in the security sense) are identified at the granularity of application messages. While this specification achieves the desired level of granularity, this approach also suffers from a few disadvantages: first, every application protocol requiring security support must include these security attributes in its message definitions; second, every application that uses the protocol must be trusted to supply the appropriate values. The goal is to preserve the appropriate granularity of security labeling while at the same time factoring security attributes out of the protocol definitions.

Recall that the FGL specification does not depend upon, nor dictate use of, RPC; rather, any suitable IPC mechanism (sockets, shared memory, special devices, etc.) can be substituted without loss of generality in the specification. It has been shown [TSOCK] that it is possible to implement *trusted* IPC mechanisms that convey security attributes, enforce policy, preserve the appropriate granularity of security labeling, and provide application programming support. In the future it would be worthwhile to integrate the FGL system with such trusted IPC mechanisms, thus demonstrating its functionality within a more generalized security framework.

SECTION 6

POLICY DAEMON IMPLEMENTATION

The purpose of the FGL daemon is to mediate access to FGL files based on the security attributes associated with the access request and the security attributes of the portion of the FGL file being accessed. As noted in the protocol description, each request sent to the daemon includes, as part of the request, a security information structure. This structure contains the security attributes of the process on whose behalf the request is being made. The association of security information with FGL file portions is described in the next subsection. How the mediation actually occurs, and how the security information associated with the daemon's response to each request is determined, is described in the subsequent subsection.

FILE SECURITY ATTRIBUTES

The description of how security attributes are stored with files is broken into two parts: 1) the association approach, and 2) the actual implementation.

Approach

There exist two fundamental approaches to retrofitting an association of security attributes with portions of individual UNIX files. One method is to use what are commonly referred to as "shadow files." In this approach, for each base file for which such an association is desired, a parallel file containing the security attributes is created. This parallel file is typically inaccessible by, and invisible to, normal users. As the base file is accessed, the shadow file is consulted in order to determine the attributes associated with the accessed portion of the base file. A second approach is to simply embed the security attributes within the base file itself. As the operating system accesses the base file, it must use and maintain these attributes, as well as hide them from applications. While the two approaches are quite similar, the following differences warrant consideration.

> *Enforcement.* The shadow file method has historically been favored in cases where a limited set of operations provide access to the base file mediated using the associated security attributes, and the remaining operations permit direct access to the base file. Clearly, if direct unmediated access to the base file is permitted, embedding security attributes in the base file itself would permit modification of those attributes in violation of the desired security policies. However, because the FGL mechanism described herein mediates *all* accesses to FGL files, this distinction is not pertinent.
>
> *Backward Compatibility.* In cases where direct unmediated access to the base file is permitted, the presence of embedded attributes would likely confound, and cause to fail, many existing applications. Shadow file implementations do not suffer this problem. The FGL mechanism described herein avoids the problem by mediating *all* accesses. In both approaches, trusted applications that wish to access the associated security information (e.g., a trusted backup utility), may easily do so.

Performance. The two approaches differ somewhat in terms of performance, including both resource utilization and speed of access. With respect to resource utilization, use of shadow files doubles the required number of files. Furthermore, in implementations where the shadow files are stored as part of the native file systems, the potential for file name collisions arises. Finally, all other things being equal, use of shadow files may detrimentally impact access speed due to the need to open, access, and close two separate files rather than one.

Review of the above considerations indicates that, in the context of the prototype FGL implementation, the two approaches are functionally equivalent. For reasons of ease of implementation, and less importantly performance, the method chosen for the FGL implementation was to embed the associated security attributes in the base file.

Implementation

The implementation for embedding security information attributes (stored in sec_info structures) was designed with the goal of optimizing the most frequently used operations. Reading, seeking, and determining file length[6] were expected to be most critical, while writing, appending, and truncating were deemed to be (relatively) less important. As a result, the data structures are intended to support rapid scanning rather than ease of updating. As a practical matter, the structures used to store the necessary information in an FGL file need not match the structures used by the policy enforcement routines in the FGL daemon.

To optimize scanning operations, rather than actually embedding sec_info structures within the body of base FGL files, two tables are kept. The first table, called the map, is comprised of a list of entries. Each entry specifies a range of offsets within the base file and an index that indicates the sec_info structure that applies to that range of bytes. The entries are ordered by offsets. Note that the range offsets are kept for user data only; the map table does not maintain entries that cover the FGL data structures. The sec_info index of each entry in the map table is an index into the second table; the table of sec_info structures. Each sec_info entry specifies a complete set of security attributes (e.g., sensitivity label and information label) to be applied to the range of bytes given in the map entry that references that sec_info entry. The entries in the sec_info table are unordered and unique. The purpose of this table is merely to cut down the storage required in the base file: rather than each range specification explicitly incorporating a sec_info structure, different ranges share common sec_info structures using the sec_info table. A figure illustrating the relationship between the map table, the sec_info table, and the user data file is shown in figure 2.

Because the tables are necessarily dynamic (i.e., they change in size as user data is written, or over-written in the file), they are kept at the end of the file. In this way, all the user data in the file need not be shifted merely because one byte, in the middle of the file, was over-written with unique attributes. In order for the policy daemon to determine where the tables are stored, a header structure (called file_hdr) is maintained at the start of the file. This

[6]Determining the file length is not, in and of itself, a frequently needed operation. File length is, however, an attribute that is returned by the internal UNIX kernel *get_attr()* routine that is very frequently used.

structure contains a pointer that indicates the offset within the file where the map table (the first of the two tables) starts, and two long integers. The first integer specifies the number of entries in the map, the second specifies the number of entries in the sec_info table. Because the sec_info table immediately follows the map table, the daemon can trivially calculate the sec_info table location by adding the map table offset to the product of the number of entries in the map table and each entry's size. Figure 3 depicts the structure of an FGL file including its associated data structures. How the various structures are created, updated, and used is described in the next subsection.

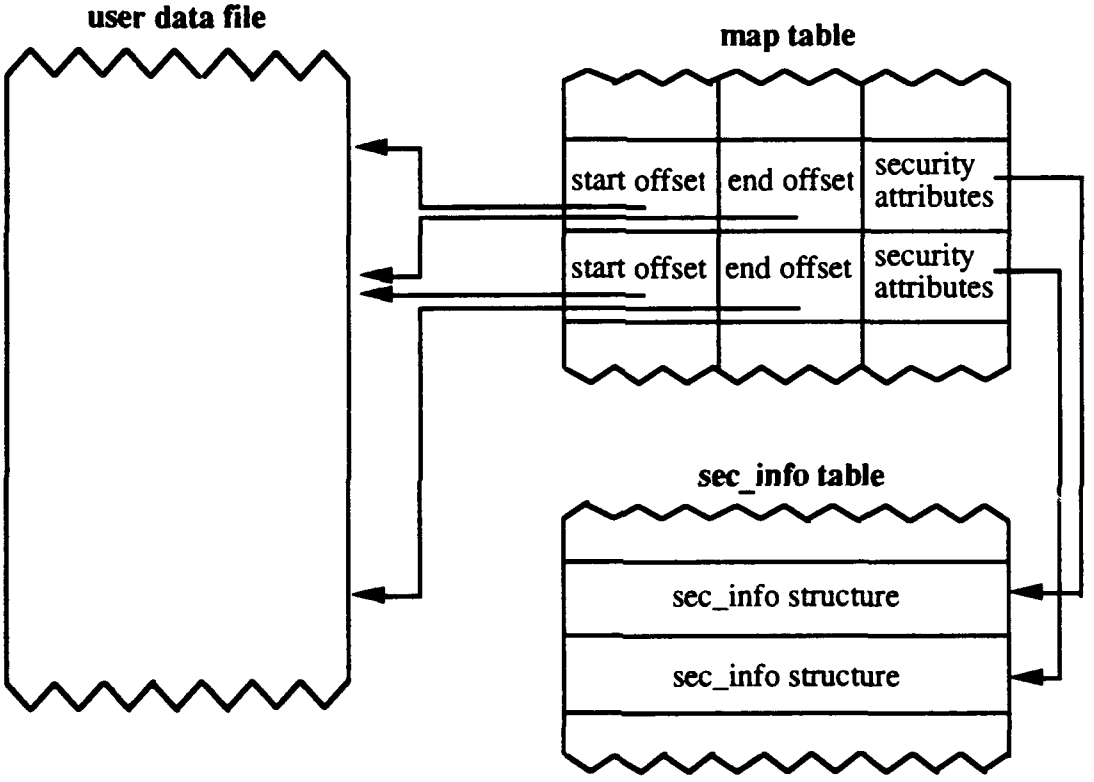


Figure 2. Data Structure Relationships

POLICY IMPLEMENTATION

As noted in previous sections, the FGL policy enforcement routines are captured in a user-level process called the FGL daemon. The daemon is a system-owned trusted process that operates at system high, but uses several privileges to bypass mandatory access control (MAC), discretionary access control (DAC), and information labeling (IL). It enforces these security policies itself by mediating access to FGL files based on the security attributes it receives with requests, and those of the portions of the FGL files being accessed. It is additionally responsible for accurately labeling its responses to the kernel.

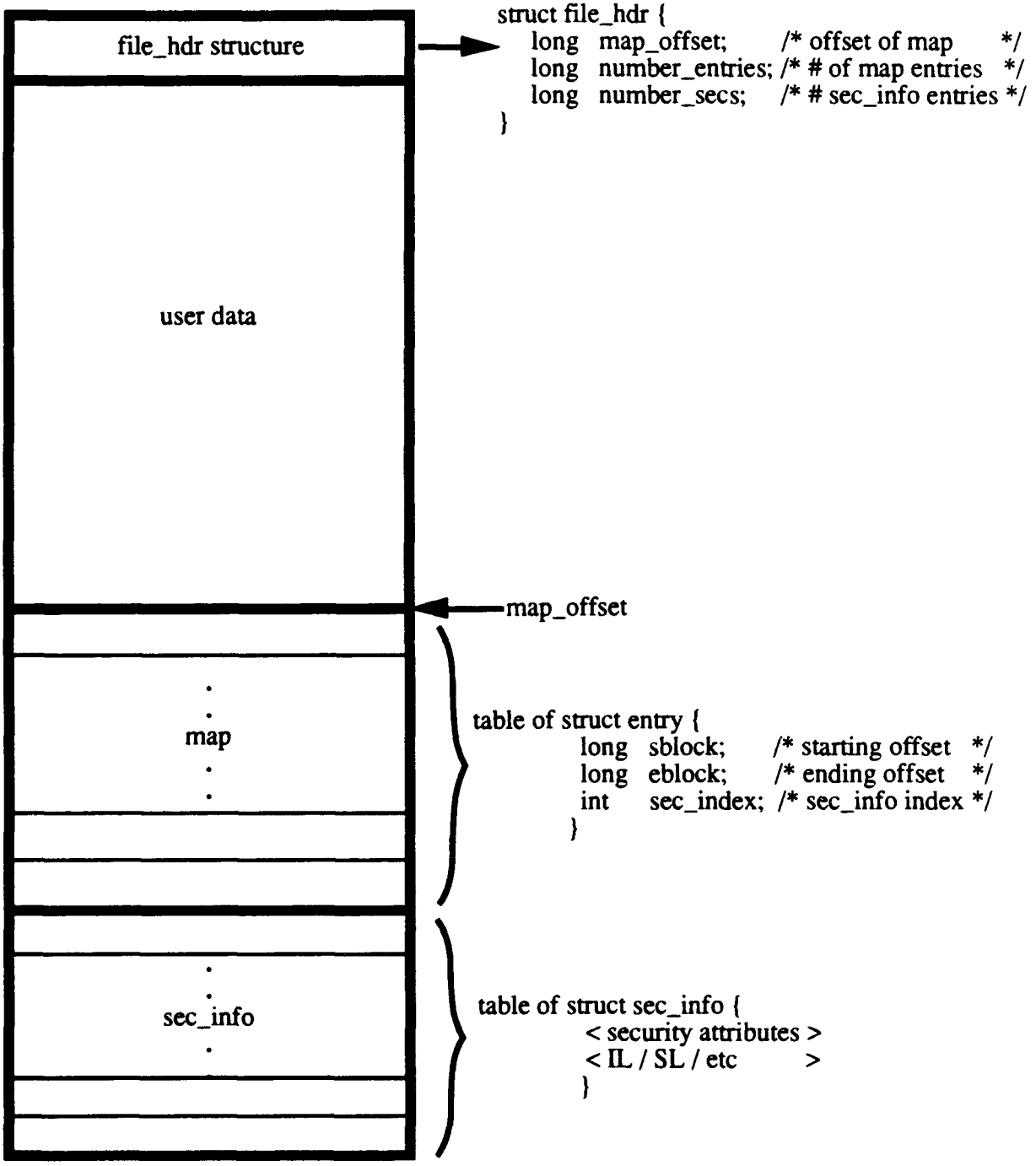


Figure 3. FGL File Structure

On start up, the FGL daemon opens a connection to a protected well-known port over which it receives FGL requests and to which it writes replies. The protocol used by the daemon was described in section 5. Once communication with the kernel has been established, the daemon enters a loop: it reads a request, dispatches based on the request type to the appropriate handler, the handler returns a response (either a positive response fulfilling the request or an error), the daemon writes the response to the kernel, and the loop repeats. The FGL daemon guarantees only that requests are processed atomically in the same order as received; as with the UNIX kernel itself, multiple processes sharing a single file must coordinate accesses among themselves or suffer potentially detrimental interactions.

As noted in the protocol section, the basic FGL functionality can be reduced to six fundamental operations: open, close, read, write, getattr, and setattr. Each of these six operations, and their associated data structures, are now described.

Open

The open request is used to open an existing FGL file. Due to the method of distinguishing FGL files from regular files, there is no means of specifying that a file should be created as an FGL file.[7] Therefore, the open request applies only to existing files.

The open request includes the absolute path name of an FGL file and a file handle unique to that file. Since the file is an FGL file, no MAC checks are necessary prior to opening the file. Because DAC remains enforced at the file level, DAC checks must be performed. For performance reasons, in this implementation the DAC checks are made in the kernel prior to sending the open request to the daemon. If the process does not have DAC access to the file, the access fails before the daemon is invoked. Thus the daemon need not perform DAC checks. Finally, information labels are not floated on opens, so no IL actions need be taken.

Although no security policy enforcement is required in the daemon on receipt of an open request, a variety of administrative actions must be performed. First, the daemon overrides MAC and DAC in order to open the file for reading and writing. Then, an entry in the FGL file table (known as mlsfiletab) is created and filled in. Each entry in this table associates an absolute path name, an open file descriptor, a file handle, a reference count, and an FGL map.

The purpose of the mlsfiletab table is to permit future requests to the daemon that reference that file to use the file handle, rather than the full path name, to reference the file. This decreases request lengths (file handles are only a few bytes, whereas path names can be several thousand bytes), and eliminates the need for the kernel to either store the full path name or to recalculate it for every request. The daemon keeps an open file descriptor as part of each entry so that it need not open, access, and close files for every request it receives. However, file descriptors are scarce resources, and the daemon may exceed the available number (typically 32 per process). For this reason, the full path name is stored. Should the daemon exceed the number of descriptors, it can close a file (e.g., using a least recently used

[7]Instead, a regular file is created, and a second system call is used to identify it as FGL. As part of the prototype, an application is provided that converts regular files to FGL files. The resulting FGL file has only one map entry whose security attributes are those that were associated with the original file.

algorithm) and reopen it, if required to satisfy future requests, using the stored path name. The FGL map portion of the mlsfiletab entry contains the FGL information (the previously described pointers and two tables) for that file. This information is read into memory at open to improve performance of future requests (i.e., so that the embedded FGL information does not need to be read from the file for each future request).

Because the daemon does not store any process state information with the open files there is only one entry in the table per file, rather than one entry per file reference. If the same FGL file is opened by multiple processes, the daemon maintains only one mlsfiletab entry, hence only one file descriptor. As a result, if a particular file is already open, the daemon does not reopen the file and create a new entry, instead it simply increments the reference count for the existing entry.[8]

Once the daemon has successfully opened the file and created the new mlsfiletab entry (if needed), it returns success to the kernel. If the file could not be opened, or the FGL information could not be read, an error is returned.

Close

The close request provides the file handle of the file to close. The daemon retrieves the mlsfiletab entry for that file handle and decrements the reference count. If the reference count remains positive, the daemon takes no further action and merely returns success to the kernel. When the reference count reaches zero, the daemon writes the FGL information back to the file (i.e., updates the file_hdr structure and writes the two tables to the end of the file), closes the file, deallocates the mlsfiletab entry, and returns success.

Read

The read request includes a file handle indicating which file to read, an offset indicating the byte offset into the file at which to start reading, the number of bytes to read, and the security attributes of the user process requesting the operation. The daemon performs two actions: first, it determines the actual offset in the context of the request sensitivity label (SL) at which to start reading, and second, it reads the number of bytes requested that are dominated by the request's SL.

On receipt of the read request, the daemon first retrieves the mlsfiletab entry and performs a linear search on the map table stored there. For each entry in the map table, the daemon looks up the entry's label (by indexing into the sec_info table), and performs a MAC check: if the request SL is dominated by, but not equal to, the entry SL then the entry is silently ignored and the next entry is processed. Otherwise, if the remaining offset is greater than the

---

[8]Note that this reference count is redundant with the usage count maintained by the kernel (described in section 5). The kernel's usage count is a more efficient approach (since the daemon is not even invoked if the count is greater than zero), but shifts some administrative burden from the daemon to the kernel. Both approaches are valid. In the prototype, the reference count predates the kernel's usage count and remains only for historical reasons.

number of bytes which this entry describes, the remaining offset is decremented by the number of bytes in the range and the loop continues.

Once the remaining offset reaches zero, the daemon seeks to the actual offset into the file (specified by the range in the last map entry processed) and reads the requested number of bytes. This data is added to an initially empty buffer whose information label starts at system low and floats as data is added. If there were too few bytes in the range of the entry then the daemon enters a second loop. It searches for the next entry whose SL is dominated by the request SL, seeks to the actual offset for that entry, and continues to read bytes and float the buffer. This second loop continues until either the requested number of bytes have been read, or the end of the file is reached.

A new file open option, O_FGL, was created that slightly alters this behavior. When a file is opened with this option, the daemon satisfies the read request with the lesser of the number of bytes requested or the number of bytes, starting at the requested offset, having the same SL and IL. (As described in the preceding section, the kernel returns the data's label to the application requesting the read as part of the data.) Since normal MAC checks are still enforced, no privilege is required to use this option.

In either case, once the read has been satisfied the daemon returns the data read, labeled with the request's SL and the data's IL, to the kernel.

Write

The write request includes a file handle indicating which file to write, an offset indicating the byte offset into the file at which to start writing, the number of bytes to write, a buffer containing the data to write, the security attributes of the user process requesting the operation, and the open flags used by the process to open the file. The daemon performs three actions: 1) it determines the actual offset in the context of the request sensitivity label (SL) at which to start writing, 2) it writes the number of bytes requested, and 3) it updates the FGL information (file_hdr structure and the map and sec_info tables).

First, the daemon retrieves the mlsfiletab entry based on the file handle. Then, because overwriting part of an FGL file is a time consuming operation, the daemon examines the open flags used by the process requesting this operation. If the O_APPEND flag (or its equivalent) was specified, thus indicating that all writes should be appended to the end of the file, the daemon executes special purpose code. This code simply appends the data to the end of the file, and updates the FGL information. This update entails the following operations:

1. Update the map_offset field in the file_hdr structure (to account for the file's increased size).

2. Compare the request security attributes with the security attributes of the last entry in the map. If they are identical, simply increase the range of bytes covered by that entry. If they differ, add a new entry to the map, and update the number_entries field in the file_hdr structure.

3. If a new map entry was added, scan the list of existing sec_infos stored in the sec_info table. If an existing entry matches the request sec_info, use that entry as the sec_info index in the new map entry, otherwise add a new sec_info entry and update the number_secs field of the file_hdr structure.

At this point the append operation is complete and the daemon returns the number of bytes appended to the kernel.

If the operation is not an append then the daemon first locates the actual offset in the file as described for read (i.e., bytes not dominated by the request SL are silently ignored). Once the actual offset has been determined, the daemon checks to ensure that the write can proceed. Specifically, if any bytes at or after the actual offset, but before the write would be satisfied, have an SL dominated by, but not equal to, the request SL, then the write fails and no data at all is written. That is, the write must be able to succeed without overwriting lower level data. Note that higher level data is simply silently ignored.

If the write is permitted to proceed, bytes are written contiguously beginning at the actual offset. The only condition under which the bytes are not written contiguously is when higher level data is skipped.

Once the write has been performed, the FGL information is updated. For brevity, the details of how this update is performed are not included here. The following principles, however, are always enforced:

1. If a map entry has its range of bytes completely overwritten, that map entry is deleted. If no other map entry shared the sec_info structure the deleted entry used, the sec_info structure referenced by the deleted entry is also deleted.

2. If the IL and SL of data being written is not identical to the IL and SL of the map entry in whose range the write occurs, a new map entry is created for the newly written data (which may involve splitting or truncating existing entries).

3. Adjacent map entries sharing common security attributes are merged.

4. Map entries always remain ordered by offsets.

5. New sec_info entries are added only if no existing entry provides exactly the desired values.

Because adhering to the above principles in all possible cases can get rather messy, the code is confusing. It is worth recalling, however, that updating the FGL information involves only changes to in-memory data structures. As a result, compared to actually writing data to the file, these updates occur quite rapidly.

At this point the write is complete. As in the case of append, the number of bytes actually written is returned to the kernel. If an error occurred (e.g., the write could not be performed because it would entail a write down), an error is returned to the kernel.

Getattr

The getattr request includes a file handle indicating the relevant file, and the security attributes of the user process requesting the operation. The getattr request is used exclusively to return the length of an FGL file. On receipt of the request, the daemon retrieves the mlsfiletab entry based on the file handle and linearly searches the map table. For each entry in the table, if the request SL dominates the entry SL, the number of bytes covered by the entry's range is added to a running total. If the request SL does not dominate the entry SL the entry is silently ignored. Once the entire map table has been searched, the total is returned to the kernel.

Note that because the map table only covers the user data portions of the FGL file, the overhead of storing security information in these files is invisible to user processes.

Setattr

The setattr request includes a file handle indicating the relevant file, a length in bytes, and the security attributes of the user process requesting the operation. The setattr request is used exclusively to set the length of an FGL file (i.e., perform truncation). As noted in previous sections, for security reasons the FGL daemon may not delete down, and for usability reasons does not delete up. Therefore, the setattr enforces a truncate equals policy.

On receipt of the request, the daemon first retrieves the mlsfiletab entry based on the file handle then locates the actual offset in the file as described for read (i.e., bytes not dominated by the request SL are silently ignored). Once the actual offset has been determined, the daemon checks to ensure that the truncate can proceed. Specifically, if any bytes at or after the actual offset have an SL dominated by, but not equal to, the request SL, then the truncate fails and no modifications are made. That is, the truncate must be able to succeed without deleting lower level data. Higher level data is simply silently ignored.

If the operation meets the above constraint the delete occurs as follows. The FGL sets an internal end of file pointer (EOFP) to the actual offset. If the EOFP starts in the middle of a map table entry, the entry is split into two at the offset point. The FGL daemon then begins a loop. It checks the next map table entry. If the entry's SL equals the request SL then the entry is deleted. If the entry's SL is not dominated by the request SL, the entry is kept, the bytes it covers in the file are read from their current location and rewritten starting at the EOFP location, the EOFP is by the number of bytes shifted, the map table entry is updated for the bytes' offset change, and the loop continues.

The same general principles adhered to when honoring write requests apply to truncation. This may result in merging many map table entries and deleting many sec_info table entries. It should be evident that, given the (potentially large) quantity of data that could be shifted due to a low-level process performing a truncate system call, the updating of in-memory data structures are unlikely to add noticeable delays.

If the truncate completes successfully the daemon returns success to the kernel, otherwise an error is returned.

OPTIMIZATIONS

No formal performance tests were applied to the FGL mechanism to determine the degree of performance degradation that it imposes. A limited amount of empirical evidence suggests the following trends: scanning operations (e.g., reading and determining file length), seem to suffer relatively minor performance hits. Even modification operations (most notably, writing) seem to offer acceptable performance when done in moderation. Extreme cases (e.g., writing files one byte at a time, each byte being individually labeled with a different label) unsurprisingly do impose a rather tremendous overhead, and as a result, a dramatic slow down.

Two points concerning this performance degradation are worth noting. The first point is that many of the applications likely to use the mechanism (such as trusted editors, mailers, and other similar applications) depend on human interaction for their operation. As a result, the performance penalties incurred through use of the FGL mechanism are less obtrusive than might otherwise be the case.

The second point is that the implementation described above was intended to incorporate the necessary functionality with the minimal possible complexity while still striving for adequate performance. It should be understood that there exist a wealth of potential optimizations well worth exploring. Indeed, it appears that many of the techniques originally conceived for such diverse applications as database management systems and disk controllers could be beneficially applied to an FGL implementation.

For example, when an FGL file is originally opened, in addition to simply reading in the FGL tables, the daemon could construct tables of indexes to quickly retrieve portions of the FGL file at a particular SL. Or, a table could be built to help rapidly perform transformations from user process-specified offsets to actual offsets. This would eliminate the need to constantly perform linear searches through the map table entries.

Similar enhancements can be made using caches. In particular, in many cases the implementation described requires that the daemon fully complete an operation prior to returning a response to the kernel. This may not always be necessary. For example, once the daemon has determined that a write can be performed, it could return the response to the kernel, then perform the write and update its maps. Similarly, the bulk of the computation required to process truncate requests need not be performed prior to returning a response. Such optimizations may require that the daemon maintain more state information pertaining to files, which might introduce an unwanted degree of complexity.

Although this prototype effort did not attempt to implement, nor even deeply consider, potential optimizations, it is worth realizing that many such enhancements exist that would likely boost performance substantially over that realized by this prototype.

CRASH RECOVERY

The ability of the FGL daemon to elegantly recover from crashes is directly related to the amount of state information it maintains and the degree to which that information can be reconstructed. As previously described, the FGL daemon does not store any process state information. Its only internal state is maintained in the mlsfiletab. In the event that the daemon fails and is restarted, any existing entries in this table will be irrecoverably lost. However, as noted in the protocol description, when the FGL daemon receives a request based on a file handle of which it has no record, it merely returns an error value indicating a stale file handle was used. The kernel then sends the appropriate request (i.e., open) to refresh that handle. The only noticeable effect of such a failure and recovery is a delay in satisfying the original user process request.

Unfortunately, the daemon as described above, suffers from a far worse difficulty, namely that it maintains, in memory, the updated FGL information (the file_hdr structure, the map table, and the sec_info table). As FGL files are modified, the daemon updates the user data portion of FGL files on disk, but only updates the memory version of the FGL information.[9] In the event of a catastrophic daemon failure, the updated FGL information would be lost. This could result in inconsistencies between the user data stored on disk and the FGL information stored on disk. Several solutions to this problem exist. One possibility is to maintain the FGL information in memory in the form of a write-through cache. An alternative is to maintain the user data updates in memory in the same fashion as the FGL information. In this way, if the daemon fails, the on-disk FGL file remains in a consistent, if somewhat aged, form. Periodic updates could be used to flush daemon caches.

Although the prototype effort has not pursued extensive efforts to address crash recovery issues, the architecture described does support such functionality. In fact, the prototype implementation could achieve this goal with only moderate updates to the daemon caching mechanism, although further research would be necessary to determine the most efficient and reliable means of doing so.

[9]The data is updated to disk to more easily support multiple processes accessing the same file. As noted in the preceding subsection, performance could be enhanced by caching such updates. The FGL information is not updated to disk for precisely this reason.

This page is blank.

SECTION 7

USE BY APPLICATIONS

A variety of applications will rely on the FGL mechanism. These applications are not limited to trusted applications designed around the FGL functionality: other applications, including untrusted applications (both those that are FGL-cognizant and those that are not), will also use FGL files. This section outlines some of the issues that must be addressed in order to support such applications.

UNTRUSTED APPLICATIONS

A large number of existing untrusted applications will successfully, yet unwittingly, use the FGL mechanism. This is because the application-level interfaces to FGL files do not differ from existing interfaces. Examples of the use of such applications include (but are not limited to):

1. Use of the UNIX **ls** command to list the contents of a directory. The file lengths reported by the command, rather than being the total number of bytes in the file, will instead be the FGL file lengths calculated based on the actual number of bytes of user data dominated by the process sensitivity level. **ls** commands performed at different sensitivity levels may yield different, correct, results depending on the labels on the portions of the FGL files.

2. Use of the UNIX **more** or **cat** commands to display an FGL file. These commands will cause only those portions of the file dominated by the process sensitivity level to be displayed.

3. Use of the **emacs** or **vi** editors to edit FGL files. These applications will allow users to view, edit, and save only those portions of the FGL files to which they have MAC access. Those potions for which MAC access is denied can neither be viewed nor modified.

In addition, untrusted applications that are FGL-cognizant can be written to provide extra functionality. Although none were developed as part of this project, examples are easily conceived. One simple FGL-cognizant application would be the UNIX **wc** program modified to report the number of lines in a document at each sensitivity level. For example, it may report that a document contains 20 lines labeled Unclassified, 10000 lines labeled Secret, and 200 lines labeled Top Secret. Based in part on this information, an analyst may decide that the lowest level at which the document could reasonably be sanitized and released is Secret. While **wc** would be expected to operate correctly, it would need to be neither trusted nor scrutinized. Furthermore, the quantity and complexity of code required to implement such functionality is minimal. The ease of implementing such functionality (indeed, the very fact that it can be implemented) superbly illustrates the intended use of the FGL mechanism.

Unfortunately, the use of untrusted applications to access and modify FGL files can lead to confusing, and unintended, results. Two problems stand out. The first is that an untrusted application, by definition, cannot maintain the individually labeled portions of an FGL file while the data is stored in the application's memory. As a result, as FGL files are read and written by untrusted applications, the utility of the FGL mechanism decreases. Consider, for example, an FGL file containing data having a Secret sensitivity level and information labels of Secret, Confidential, and Unclassified. When a process running at Secret reads all the data and rewrites it to the FGL file (even without modifying the data), the entire FGL file will be labeled with both a sensitivity and information label of Secret. Such floating, though inevitable when using untrusted applications, runs counter to the goal of the FGL mechanism and is therefore undesirable.

A worse problem is the confusion that can be caused due to the hidden higher level data present in an FGL file. Consider an FGL file containing data having sensitivity levels Secret and Unclassified. If an application with a sensitivity level of Unclassified reads the file, truncates it to half size, and writes back some portion of the Unclassified data, the Secret data (which remains in the file throughout) is likely to no longer maintain its intended position, relative to the Unclassified data. (For example, if the file were a document containing a Secret comment about an Unclassified paragraph, the two paragraphs may end up shifted relative to each other.) This type of shifting, which is also inevitable, can be highly confusing and extremely annoying, and could potentially alter or destroy the intended semantics of the stored data.

Finally, and most problematically, some untrusted applications that unwittingly use FGL files will simply fail. The reason is the inability of such applications to delete data visible to them in a file. For example, a Secret application may read an FGL file containing data having a sensitivity level of Unclassified. If the application then attempts to overwrite that data, or truncate the file to eliminate the data, the request will fail and the application will receive an error (e.g., a mail program running at secret may attempt to delete an Unclassified message once the user has read it). Because the error received is the result of the slightly new semantics imposed by the FGL nature of files, unmodified applications will probably be unable to recover from the error.

Thus, although many benefits can be realized using FGL files, even with untrusted applications, great care must be taken to ensure that FGL files can successfully be manipulated by those applications, and that such manipulations do not cause unpredicted and confusing results.

TRUSTED APPLICATIONS

The mechanism described in this document provides very little support for trusted applications. The O_FGL open option permits applications to cause read requests to be constrained by label change boundaries and to return the label of the data actually read, and the daemon does support the base CMW's MAC and IL override privileges. In addition, as a last resort, trusted applications may bypass the FGL daemon and directly access FGL files and the stored FGL information. However, the FGL mechanism probably requires far more

sophisticated and convenient interfaces to adequately support trusted application development.

Some applications may need interfaces to easily determine the labels associated with particular portions of the file (useful to a trusted editor, for example), while others may need the ability to easily modify the attributes associated with portions of a file (for example, a trusted database management system may need to directly modify security attributes).

Specifying an appropriate set of application level interfaces, the privileges that should be required for those interfaces, and the side-effects of those interfaces is not a trivial problem. Because the FGL project described here focussed on the underlying mechanism, this area remains ripe for future research.

PORTABLE TRUSTED APPLICATIONS

There are two aspects to developing portable trusted applications that rely on an FGL mechanism. Ideally, the semantics of FGL files and a complete application-level interface specification should be specified and adopted as an industry standard. This would provide the best long term solution, not just for portability of trusted applications, but for all applications dependent upon the FGL mechanism.

While interoperability and portability through standardization may eventually be achieved, short term solutions exist that satisfy the need for portability of trusted applications. One solution (implemented as part of this prototype) is to replace the FGL daemon's RPC-based front end with a library interface. Because the FGL policy enforcement is then library-based, applications which do not use the FGL library cannot be permitted to gain access to the file (otherwise the embedded FGL information is jeopardized). Furthermore, sharing common FGL files requires even more coordination (since the FGL libraries would not provide coordination among themselves). However, for portable trusted applications that require the FGL functionality on their own, privately kept files, such an approach is quite adequate. Furthermore, this approach allows the basic FGL labeling functionality to be implemented on arbitrary systems: CMWs, system high hosts, even unmodified commercial UNIX systems all can support such applications.

Applications cannot be retrofitted with "portability"; any application is portable to some set of systems. The size of that set depends on the assumptions made by the programmer. In order to head towards application portability for FGL-based applications, a minimal set of interfaces need to be defined. On existing systems that do not inherently support FGL files, these interfaces can be implemented as library routines. As the functionality becomes more accepted, and availability becomes more widespread, the interfaces will become part of base operating systems (e.g., as system calls in UNIX).

For the sake of portability and interoperability of future applications, it is therefore important that the application-level issues associated with FGL mechanisms be explored and that a minimal interface specification be defined. As noted above, thus far this area remains uncharted territory.

This page is blank.

SECTION 8

SUMMARY AND CONCLUSIONS

Trusted systems that naively implement restrictions in order to meet the CMW requirements or the TCSEC B-level criteria cause certain applications to cease functioning properly. These failures are primarily caused by unsophisticated implementations of mandatory access control policies. More specifically, because sensitivity labels and mandatory access control are applied on a per-file basis, applications that expect to be able to use well-known shared files now fail when those files are shared across sensitivity levels. There are many important applications of this type. Existing trusted systems have addressed selected parts of the problem by implementing awkward solutions of little general applicability. The fine grained labeling (FGL) mechanism presented in this paper is intended to provide a complete and general solution to the labeling granularity problem.

The FGL design and prototype effort has demonstrated several points. Most importantly, the prototype has shown that it is possible to implement a fine grained labeling policy that provides labeling and access control down to the byte level. This degree of granularity not only solves the problems associated with file-level labeling and access control, but also provides a mechanism on which to implement both trusted and untrusted security-cognizant applications that provide enhanced labeling features.

The prototype has also shown that the impact of retrofitting an FGL mechanism to an existing UNIX-based CMW is not unbearable. The implementation described in this paper required neither syntactic nor semantic modifications to the UNIX interface to the file system. Thus, binary level backward compatibility with existing applications can be assured. In addition, by using a daemon-based architecture the prototype implementation required only a small amount of code to be embedded in the kernel; the bulk of the mechanism was implemented as a user-level process. As a result, system developers wishing to incorporate FGL functionality into future versions of their current products would not need to make extensive changes to their existing software.

Given the daemon architecture, an FGL protocol is required to communicate requests and responses to and from the daemon. The FGL protocol requires only a few, relatively simple, messages. Once the protocol syntax and semantics have been established FGL daemons can be written and easily ported between machines. So, although the FGL daemon has proven to be a complex and rather large application, once one implementation has been developed, that code can easily be shared among multiple systems, even multiple vendors, if desired.

The performance impact of the mechanism presented in this document was not formally analyzed, however, the mechanism appears to achieve adequate performance despite little emphasis being placed on efficiency or optimization. Many possible methods of increasing performance without detracting from the functionality were proposed. Some of these optimizations can reasonably be expected to dramatically increase performance by eliminating bottlenecks. It is quite likely, therefore, that very good performance could be achieved in a more finely tuned implementation.

The impact on existing applications was not studied in great detail, however, it is immediately obvious that untrusted FGL-ignorant applications can benefit from the existence of the mechanism. Indeed, these are the very applications that fail with file-level labeling policies that the mechanism is intended to support. Unfortunately, it is also clear that, in some cases, untrusted FGL-ignorant applications will produce unexpected and confusing results. Therefore, wholesale replacement of existing file-level labeling policies with FGL policies is not recommended.

As a long-term solution to the problems stemming from file-level labeling, the FGL mechanism holds much promise. A variety of FGL-cognizant untrusted applications can be easily written that provide additional security-related functionality. Furthermore, the development, porting, and evaluation of trusted applications that require fine grained labeling to provide enhanced security features will all be greatly simplified through the use of a common operating system-supported FGL mechanism.

At this time, use of the FGL mechanism by trusted applications remains very poorly understood. The semantics of privileged interfaces, the appropriate syntax for such interfaces, and other application-level issues remain completely unexplored. Related human factors issues, such as how trusted applications that use the FGL mechanism should present the additional security information to users, also remain unexplored.

Therefore, while the prototype has successfully demonstrated the feasibility and practicality of a fine grained labeling mechanism, future research is desperately needed to investigate how best to capitalize on the enhanced trusted system capabilities that such a mechanism provides.

LIST OF REFERENCES

[BELL] Bell, D. E., La Padula, L. J., "Secure Computer Systems: Unified Exposition and Multics Interpretation," MTR 2997, The MITRE Corporation, July 1975.

[CMWPROTO] Berger, J. L., Picciotto, J., Woodward, J. P. L., Cummings, P. T., *Compartmented Mode Workstation: Prototype Highlights*, IEEE Transactions on Software Engineering, June 1990, pp. 608-618.

[CMWREQS] Woodward, J. P. L., "Security Requirements for System High and Compartmented Mode Workstation," MTR 9992, Revision 1, The MITRE Corporation, November 1987 (also published by the Defense Intelligence Agency as document DDS-2600-5502-87).

[OBJTYPE] Graubart, R. D., Picciotto, J., Fohlin, J. C., "Using Object Typing to Support Trusted Applications," M91-84, The MITRE Corporation, 1991

[RPC] "RPC: Remote Procedure Call Protocol Specification, Version 2," Sun Microsystems, Inc., June 1988, (RFC 1057, Network Information Center, SRI International).

[SECPOSIX] "Draft Security Interface for the Portable Operating System Interface for Computer Environments," IEEE P1003.6 Draft 12, September 1991.

[TCSEC] "Department of Defense Trusted Computer System Evaluation Criteria," DOD 5200.28-STD, December, 1985.

[TSOCK] Vukelich, D. F., "Trusted X Window System, Volume 3: Client/Server Communication," to be published as MTP 288 Volume 3, The MITRE Corporation, 1991.

[XDR] "XDR: External Data Representation Standard," Sun Microsystems, Inc., June 1987, (RFC 1014, Network Information Center, SRI International).

This page is blank.

APPENDIX

FGL PROTOCOL SPECIFICATION

This appendix provides the protocol definition, in XDR notation, of the FGL protocol used to communicate between the kernel and the FGL daemon. Section 5 of this document provides a high level description of this protocol and how it is used in the prototype FGL implementation.

Interface Definition

```
program FGLPROG {

    version FGLVERSION {

        fglstat
        FGLPROC_OPEN (fglopenargs) = 1;

        fglstat
        FGLPROC_WRITE (fglwriteargs) = 2;

        fglrdresult
        FGLPROC_READ (fglreadargs) = 3;

        fglattrstat
        FGLPROC_GETATTR (getattrargs) = 4;

        fglstat
        FGLPROC_SETATTR (setattrargs) = 5;

        fglstat
        FGLPROC_CLOSE (fglfhandle) = 6;

    } = 1;
} = 44;
```

Constants

Note: Sizes are given in decimal bytes.

```
FGL_MAXDATA       2048   /* Maximum for data read/written per request */
FGL_MAXNAMLEN     255    /* Maximum for a component of a path name */
FGL_MAXPATHLEN    1024   /* Maximum path name length */
FGL_PORT          2051   /* Port number used for contacting FGL server */
FGL_FHSIZE        16     /* Size of file handle */
FGL_SASIZE        64     /* Size of security attributes */
```

RPC retransmission parameters

```
FGL_TIMEO       7       /* Initial timeout in tenths of seconds. */
FGL_RETRIES     3       /* Times to retry request. */
```

General Data Types

The following XDR definitions are basic structures and types used in the FGL protocol.

fglstat

Note: The error values are derived from UNIX error numbers.

```
enum fglstat {
    FGL_OK = 0,                 /* no error */
    FGLERR_NOENT=2,             /* No such file*/
    FGLERR_IO=5,                /* No such file*/
    FGLERR_STALE=70             /* Stale FGL file handle */
};
```

FGL_OK
A value of FGL_OK indicates that the call completed successfully and that the result parameters, if any, are valid.

FGLERR_NOENT
The file or directory specified does not exist.

FGLERR_IO
The file or directory specified does not exist.

FGLERR_STALE
The file handle (see below) given in the request arguments was invalid. That is, the file referred to by the file handle supplied either no longer exists or access to the file has been revoked.

fglfhandle

```
typedef opaque fglfhandle[FGL_FHSIZE];  /* File Handle */
```

This is the file handle, which is used as a shorthand for absolute pathnames in all FGL procedures.

fglpathname

```
typedef string fglpathname[FGL_MAXPATHLEN];
```

A pathname in FGL is a full (i.e., absolute) pathname.

fglfattr

```
struct fglfattr {
    u_long          na_size;        /* file size in bytes */
};
```

This structure is a set of file attributes. Currently, the only file attribute that the daemon must manage is file size.

sec_info

```
typedef opaque sec_info [FGL_SASIZE];
```

These are the security attributes that accompany the requests/responses and are used to enforce access policies on subjects/objects.

fglattrargs

```
struct fglattrargs {
    fglfattr    ns_attr_u;
    sec_info    ns_si;
};
```

This structure is a set of file attributes protected by a set of associated security attributes; it is considered an object (in the security sense) of the FGL system.

FGL Procedures

Open File

```
fglstat
FGLPROC_OPEN (fglopenargs) = 1;

struct fglopenargs {
    fglfhandle      fh;
    sec_info        si;
    fglpathname     path;
};
```

Write File

```
fglstat
FGLPROC_WRITE (fglwriteargs) = 2;

struct fglwriteargs {
    fglfhandle      wa_fhandle;   /* handle for file */
    sec_info        wa_si;
    u_long          wa_offset;    /* current byte offset in file */
    u_long          wa_flag;
    u_long          wa_count;     /* size of this write */
    char            *wa_data;
};
```

Read File

```
fglrdresult
FGLPROC_READ (fglreadargs) = 3;

struct fglreadargs {
    fglfhandle      ra_fhandle;   /* handle for file */
    sec_info        ra_si;
    u_long          ra_offset;    /* byte offset in file */
    u_long          ra_flag;
    u_long          ra_count;     /* read count */
};

struct fglrrok {
    sec_info        rrok_si;
    u_long          rrok_count;   /* bytes of data actually read */
    char            *rrok_data;
};

union fglrdresult switch (fglstat rr_status) {
    case FGL_OK:
        struct fglrrok  rr_ok_u;
    default:
        void attrs;
};
```

Get File Attributes

```
fglattrstat
FGLPROC_GETATTR (fglgetattrargs) = 4;

struct fglgetattrargs {
    fglfhandle ga_fh;
    sec_info   ga_si;
};

union fglattrstat switch (fglstat at_status) {
    case FGL_OK:
        fglattrargs fa;  /* file attributes object valid */
    default:
        void attrs;
};
```

Set File Attributes

```
fglstat
MLSPROC_SETATTR (fglsetattrargs) = 5;

struct fglsetattrargs {
    fglfhandle sa_fh;
    fglattrargs sa_fa;
};
```

Close File

```
fglstat
MLSPROC_CLOSE (fglfhandle) = 6
```

This page is blank.

DISTRIBUTION LIST

INTERNAL

A030

R. W. Jacobus
H. W. Sorenson

D010

E. J. Ferrari, Jr.

D040

G. J. Koehr

D042

J. R. Boonstra II

D050

R. A. Games
R. A. McCown

D060

D. D. Neuman

D070

D. A. MacQueen

D080

C. H. Cager

D090

L. S. Metzger

D100

S. J. Pomponi

G010

E. L. Burke
V. A. DeMarines
C. M. Sheehan

G030

R. F. Nesbit

G034

D. H. Lehman

G110

H. A. Bayard
E. L. Lafferty
L. J. La Padula
J. K. Millen
P. S. Tasker

G111

R. H. Adams
S. L. Bayer
J. L. Berger
M. P. Chase
J. C. Fohlin
B. A. Goodman
E. J. Green
D. W. Lambert
J. E. LeMoine
R. A. Marcotte
M. T. Maybury
B. O'Neill
J. Picciotto (15)
R. L. Ryan, Jr.
A. M. Tallant
M. B. Vilain
D. F. Vukelich (10)
J. S. Wood
T. J. Woodhouse
J. P. L. Woodward

G113

D. A. Chodorow
J. Kowalchuk, III
M. Krajewski, Jr.
A. B. Murphy
M. S. Panock

G115

S. R. Ames, Jr.
D. L. Baldauf
E. M. Caddick
M. H. Cheheyl

G116

F. Belvin
N. D. Buchanan
D. A. Conte
M. Ferdman
J. W. Francis
G. J. Gagnon
W. R. Gerhart
R. D. Graubart
P. J. Guay
S. T. Kawamoto
C. J. Murphy
J. F. Myers
R. H. Walzer

G117

D. J. Bodeau
T. J. Brando
R. K. Burns
F. N. Chase
W. M. Farmer
H. G. Goldman
J. D. Guttman
D. M. Johnson
R. C. Labonte
L. G. Monk
M. E. Nadel
J. D. Ramsdell
B. M. Thuraisingham
M. L. Urban

G117 (continued)

J. G. Williams
M. M. Zuk

G140

M. D. Abrams
J. H. Babcock
S. Jajodia
W. H. Neugent
J. V. Patton
R. W. Shirey

G143

J. T. Pirocchi

G147

L. J. Fraim
G. A. Huber
A. B. Jeng
P. W. Mallet
C. McCullom
L. A. Noble
L. Notargiacomo
C. Paradise
M. E. Rudell
S. I. Schaen
L. M. Schlipper
J. M. Vasak
D. M. Venese

G148

D. L. Martell
W. P. Nenninger

H024

S. Chokhani
D. L. Gill
W. N. Havener
K. C. Rogers
R. L. Williamson